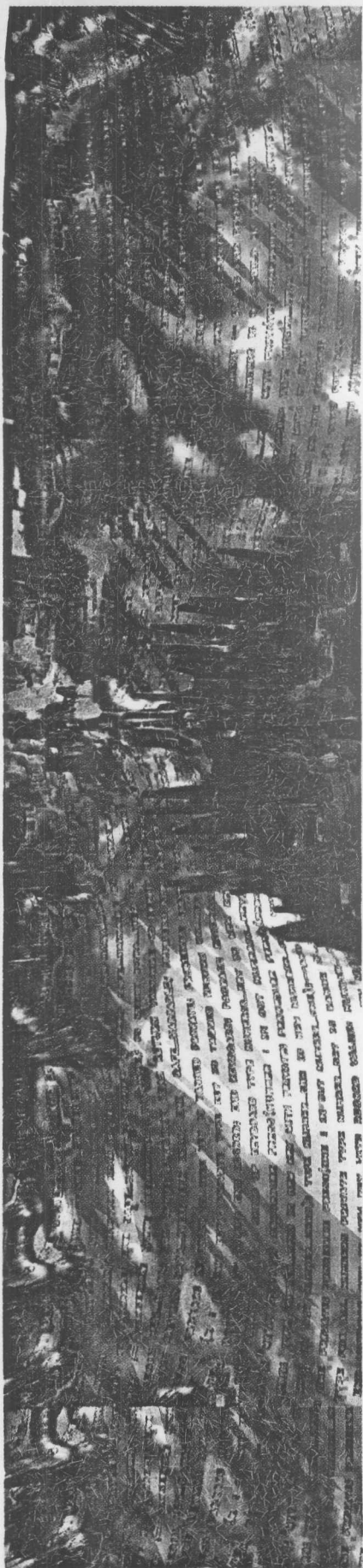




Skeleton Stoodios

How to Write a PID Algorithm

The tricky part is trying to build a PID that operates smoothly in all modes of operation.

When you hear of people you know having an angiogram or stents put in a collapsed artery, you realize these procedures can involve a whole variety of medical catheters of differing sizes. These catheters are manufactured by the process of polymer extrusion. Quality control of the catheters demand a precise outside diameter with tolerances of ± 0.0005 in. One method of maintaining an accurate outside diameter is to maintain accurate polymer pressure inside the extruder. The need for precise pressure control is what brought me to the task of writing my own PID (proportional-integral-differential) algorithm. Those of you who have worked with PID controllers know that a wide variety of algorithms exist in the industry. The intent of this article is not to cover all PID algorithms but to show one implementation of a "home-made" PID and how I managed some of the trickier aspects of PIDs, such as bumpless transfer and setpoint changes.

The extrusion systems I work on use a Delta Tau Data Systems PMAC (programmable multi-axis controller) to control a servo motor that drives the extrusion screw that in turn provides the polymer pressure. In order to control pressure, the PMAC A/D converter reads the melt pressure and then passes data to my pressure PID whose output is then cascaded into the setpoint of the extruder screw servo motor control loop. My PID is written in the PMAC motion control language, which is similar to Basic, and the code

examples can be easily converted to C, C++, or other languages.

Up until writing this algorithm, I had always depended on PIDs supplied with the control hardware/software package I was using. Whether I was working on a PLC, a motion controller, or a PC-based control system, I was only required to insert the PID into the control logic, run the program, and tune the loop. As I said earlier, this is an extruder pressure control loop whose output is cascaded into the setpoint of a screw servo loop. One option I had was to employ an unused axis of the 8-axis motion controller for this pressure loop and cascade its output into the screw loop. This option would have wasted the motion control axis which might be needed later, so I decided to write my own loop and embed it in the screw servo motion program. Each time the servo motion program executed, the pressure loop would also execute and deliver a new screw loop setpoint.

I started researching the subject and found many books on the subject of PIDs. Most were heavy on theory and either light or weightless on practical "show-me-the-code" kind of information I was looking for. Control system integrators I talked to knew how to tune one but not how to write one. It was black magic. I then found a book on the basics called *Real-Time Computer Control: An Introduction* (Stuart Bennett, Prentice Hall International, Hertfordshire, UK, 1994). The book covered writing a generic PID loop and also included some methods of bumpless transfer. I put together a basic PID and had the

Until writing this algorithm, I had always depended on PIDs supplied with the control hardware/software package I was using.

folks at Delta Tau check my algorithm. They gave it a thumbs up but mentioned that I might have difficulty going from manual to auto and back without further code. I was so excited to have something that was close to implementation that I said I would figure it out and headed for the extrusion lab. As soon as I ran some test numbers through the algorithm I knew that they weren't kidding. More code was needed. I didn't like the bumpless transfer methods in the book, and after banging my head against the extruder for a few hours, I came up with the method described in the examples.

Many differing PID algorithms are used in industry. Each PLC manufacturer that offers a PID adds a custom twist to the algorithm. Each PC-based control system that has a PID also adds a custom twist. The only exception is the classic ISA PID algorithm that conforms to a standard. For the purposes of my examples below, I will be referring to the *independent* or parallel PID algorithm (which I used). See the sidebar "Abbreviations" for a few definitions of the abbreviations I use in the following text and code fragments.

ERROR

As its name implies, a PID consists of three components or terms: proportional, integral, and derivative. The proportional term is a product of error and proportional gain. Error is calculated by finding the

difference between the setpoint and the process variable. This can be done by either:

$$\text{Error} = \text{SP} - \text{PV}$$

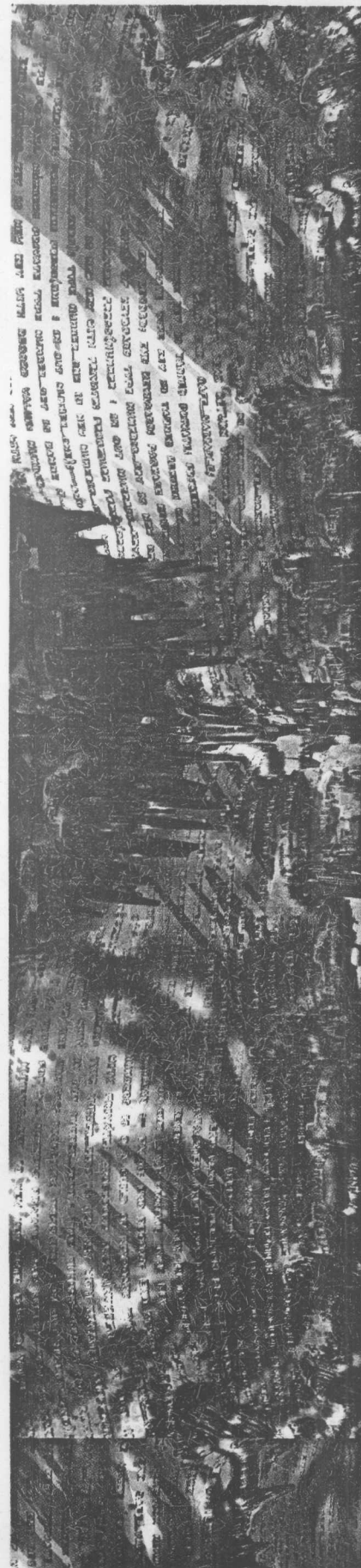
or:

$$\text{Error} = \text{PV} - \text{SP}$$

The method you select will determine whether your control loop is a forward or reverse acting PID (some call this either inc/inc or inc/dec). In other words, as the output of the loop increases, does the process variable go up or go down? How your process responds will determine which method you choose. For example, I used the first method above of calculating error in my polymer pressure loop PID. An increase in the output of that loop (servo velocity command) turns the screw faster, which in turn yields a higher pressure. Another way to look at it is if there is positive error in Equation 2, the pressure is lower than the setpoint. Positive error will always yield

ABBREVIATIONS

- Setpoint (SP)
- Process Variable(PV)
- Proportional Gain (KP)
- Integral Gain (KI)
- Derivative Gain (KD)
- PV High Limit - PV Low Limit (PV_Span)
- CV High Limit - CV Low Limit (CV_Span)
- PID or Manual Flag (Auto_Mode)
- Error (Error)
- Sum of the Error (Sum_Error)
- Change in Process Variable Since Last PID Execution (d_PV)
- Change in Error Since Last PID Execution (d_Error)
- Proportional Term (PTerm)
- Integral Term (ITerm)
- Derivative Term (DTerm)
- Process Variable on Last PID Execution (Last_PV)
- Error on Last PID Execution (Last_Error)
- Error when manual to auto (Initial_Error)
- Last Setpoint when setpoint changes (Last_SP)
- Controlled Variable (CV)
- Manual RPM Setpoint (Manual_Command)
- PID Execution Stage (First_PID_Execution)

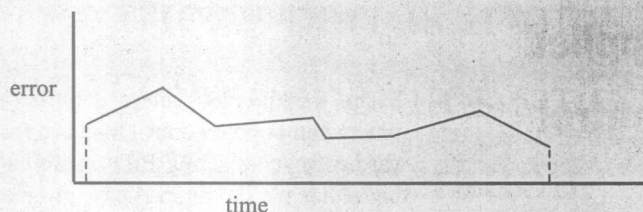


PID Algorithm

an increase in the loop output (we'll see that later), which, in this case, will turn the screw faster and increase the pressure, thus reducing the error. If I had chosen the second method and there had been any error, the loop would have sent the screw velocity in the wrong direction (faster for high pressure or slower for low pressure) and the pressure would have "run away" in the direction of the error.

An example of using the method in Equation 2 is a tank level control using a proportional valve to let liquid out as an amount of liquid is being pumped in. In this case, the PV is the actual level and the SP is the desired level. The output of the loop controls the percent the discharge valve is open, from 0% (fully closed) to 100% (fully open). After a bit of thought it is apparent that Equation 2 is the best choice. Let's do a what-if to make sure. If the tank were too low, it would cause negative error,

FIGURE 1
Error over time.



which would then reduce the controller output, closing the valve, and bring the level back up. If Equation 1 were chosen, a low tank would cause positive error, open the valve more, and empty the tank.

PROPORTIONAL TERM

The proportional term is so named because its output is proportional to error. It responds to instantaneous error, error

that is calculated on that one execution of the PID. If you had a P-only controller, with a K_p greater than zero, the controller's output will increase as the error increases and decrease as error decreases. The typical (simplified) code fragment for a P-only controller is:

$$P_{term} = K_p * Error$$

$$CV = P_{term}$$

PRECISE SOLUTION 2.0

The Royalty-Free Alternative For Advanced Real-Time Applications

An integrated development environment for real-time embedded design, which includes:

- Precise/MQX Real-Time Executive
- Precise Embedded I/O Components:
 - TCP/IP
 - LAPB
 - PPP
 - LAPD
 - RPC
 - HDLC
 - CAN
 - SNMP
- Design Tool
- Performance Tool
- Task-aware debugging
- Compilers

Precise/MQX Real-Time Executives

ColdFIRE, 68K, 683xx

Power PC 401, 403, 603, 604

MPC 801, 821, 823, 860

MIPS R3000, R4000

TMS320C3x, TMS320C4x

ARM

56K, Onyx

Sales: (800) 628-8631 info@psti.com <http://www.psti.com>

CIRCLE # 34 ON READER SERVICE CARD

PID Algorithm

As you can also see, the higher the K_p the higher the output for a given amount of error. That's why a loop with a high proportional gain is more responsive to error than one with a respectively lower gain.

Those of you with a perceptive eye might have noticed something that didn't seem like it would work: units. How do you handle dissimilar input and output units? My pressure PID SP and PV units are in psi, while the CV is in RPM. The process of converting the units is called normalization and is handled by setting up a ratio of the input and output spans, and multiplying the result by the variable you want to convert (error in this case). The final Pterm code fragment is as follows:

```
Error = (SP - PV) * (CV_Span / PV_Span)
```

or:

```
Error = (PV - SP) * (CV_Span / PV_Span)
```

```
;Then
Pterm = KP * Error
CV = Pterm
```

INTEGRAL TERM

The integral term is so named because its output is the integral of the error with respect to time. Its job is to remove long term or steady state error that is not removed by the Proportional Term. As you may remember from calculus, the first integral of an equation of a line is equal to the area under the line. In Figure 1, the line shows error over time and the area under the line is the integral of the error. We calculate the integral by adding the error to itself each time the PID executes. Integral term code fragment in an integral-only controller looks like this:

```
Error = (SP - PV) * (CV_Span / PV_Span)
```

or:

```
Error = (PV - SP) * (CV_Span / PV_Span)
;Then
Sum_Error = Sum_Error + Error
Iterm = KI * Sum_Error
CV = Iterm
```

If an error persisted that was not corrected by the proportional term and the KI is greater than zero, the integral term would begin to grow larger and larger eventually correcting or "resetting" the long term error.

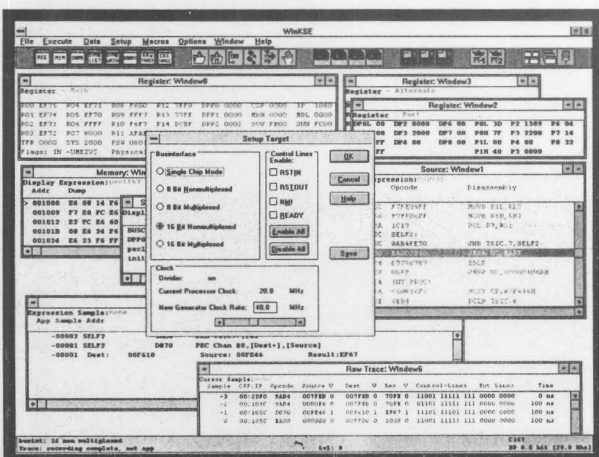
DERIVATIVE TERM

The derivative term is meant to slow down changes in the process being controlled. This slowing is done by calculating the changes in the process or the "derivative" of the process with respect to time. Two indicators of process change are available, PV and error. Derivative Term code fragment is as follows:

;PV Method

Uncompromised Visibility

Full emulation support of Siemens 80C166/167, C165, C163, VeCon and SGS Thomson ST10



Debug in any mode, including single chip mode, while viewing all internal registers, busses and on-chip peripherals.

Plus! Full support for Intel386™ and Intel486™ microprocessors

- ✓ Unique bondout technology guarantees emulation under all conditions. Run, break and trace control of all internal busses and on-chip peripherals
- ✓ Emulation support of the internal on-chip RAM and ROM areas with zero wait state operating speeds
- ✓ Display and modification of all registers including Port, Main, Special Function, and Extended Special Function
- ✓ Up to 64K trace with time stamp
- ✓ Supports all voltages and clock speeds
- ✓ User defined sequences for complex breakpoints
- ✓ Full-speed, real-time emulation with or without target hardware
- ✓ Source level debug support for all popular compilers
- ✓ Integrated MS Windows® environment

Phone: 800-566-8766
Fax: 714-851-3180
E-mail: dtsales@kontron.com
Internet: http://www.kontron.com

Newport Beach, CA
 Sales Representation Nationwide



KONTRON ELEKTRONIK

CIRCLE # 36 ON READER SERVICE CARD

PID Algorithm

LISTING 1

Screw servo motion program.

```

01      While (True)
02          If (Auto_Mode = 1 )
03              If (First_PID_Execution = 0 OR First_PID_Execution = 1)
04                  Error = (SP - PV) * (CV_Span / PV_Span)
05                  Sum_Error = Sum_Error + Error
06                  If (First_PID_Execution = 0)
07                      Sum_Error = Manual_Command / KI
08                      First_PID_Execution = 1
09                      Initial_Error = Error
10                  Endif
11                  Pterm = 0
12                  Iterm = KI * Sum_Error
13                  Dterm = 0
14                  If (Initial_Error > 0 and Error < 0)
15                      First_PID_Execution = 2
16                      Last_PV = PV
17                  Endif
18                  If (Initial_Error < 0 and Error > 0)
19                      First_PID_Execution = 2
20                      Last_PV = PV
21                  Endif
22                  Last_SP = SP
23              Else
24                  If (SP = Last_SP)
25                      Error = (SP - PV) * (CV_Span / PV_Span)
26                      Pterm = KP * Error
27                      Sum_Error = Sum_Error + Error
28                      Iterm = KI * Sum_Error
29                      d_PV = (Last_PV - PV) * (CV_Span / PV_Span)
30                      Last_PV = PV
31                      Dterm = KD * d_PV
32                  Else
33                      Error = (SP - PV) * (CV_Span / PV_Span)
34                      Sum_Error = Sum_Error + Error
35                      Pterm = 0
36                      Iterm = KI * Sum_Error
37                      Dterm = 0
38                      If (SP > Last_SP and PV > SP)
39                          Last_SP = SP
40                          Last_PV = PV
41                      Endif
42                      If (SP < Last_SP and PV < SP)
43                          Last_SP = SP
44                          Last_PV = PV
45                      Endif
46                  Endif
47              Endif
48              CV = Pterm + Iterm + Dterm
49          Else
50              CV = Manual_Command
51              First_PID_Execution = 0
52          Endif
53      EndWhile

```

```

Error = (SP - PV) * (CV_Span/PV_Span)
d_PV = (Last_PV - PV) * (CV_Span/PV_Span)
Dterm = KD * d_PV
Last_PV = PV
CV = Dterm

```

```

;Error Method
Error = (SP - PV) * (CV_Span/PV_Span)
d_Error = Last_Error - Error
Dterm = KD * d_Error
Last_Error = Error
CV = Dterm

```

As you can see, if the PV method used, the normalization code must be added in. If the error method is used, normalization has already been done in the error calculation. The output of the derivative term will be large if K_D is greater than zero and there is a large change in either PV or error. Changing the derivative gain will change the effect that the Dterm has on the final output.

PUTTING IT ALL TOGETHER

In a complete PID algorithm, all three terms are put together. The code fragment that follows is a forward acting, PV based Dterm PID:

```

Error = (SP - PV) * (CV_Span / PV_Span)
Pterm = KP * Error
Sum_Error = Sum_Error + Error
Iterm = KI * Sum_Error
d_PV = (Last_PV - PV) * (CV_Span / PV_Span)
Last_PV = PV
Dterm = KD * d_PV
CV = Pterm + Iterm + Dterm

```

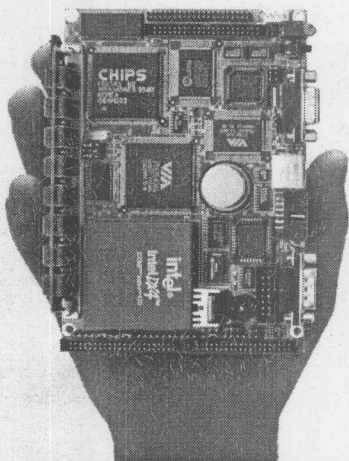
So that's it. How to write a PID. Good to it and good luck. But wait, will the thing really work? You know, in the real world? Well, yes and no. This PID would work but not transition from manually controlled process to a PID controlled process very gracefully. It would also not handle setpoint change very well. So let's take a look at the two issues. Because the remaining code will have more features, refer to Listing 1 in order to follow along in the text.

Embedded PC

Fits your applications
and budget!

Biscuit PC

PCM-4824
486 SBC with SVGA/LCD



Fully PC Compatible. Easy to Install.
Integrated Features Include FDD, HDD,
2S/1P ports, Watchdog timer, PC/104
expansion and more...



Biscuit PC, full size (5 1/4" FDD size)

PCM-5860 Pentium SBC with SVGA/LCD,
Ethernet and PCI Slot
PCM-4862 486 SBC with SVGA/LCD, Ethernet
and SDD

Biscuit PC, half size (3 1/2" HDD size)

PCM-4824 486 SBC with SVGA/LCD
PCM-4822 486 SBC with Ethernet
PCM-4820 486 SBC
PCM-3864 386 SBC with SVGA/LCD

Slot Single Board PC (ISA)

PCA-6151 Pentium half-size all-in-one CPU
card with SVGA
PCA-6145 486 half-size all-in-one CPU card
with SVGA/LCD and Ethernet

PC/104 Modules

• Solid State Disks • PCMCIA • Ethernet • VGA/LCD
• Multi-port RS-232 • RS-422/485 • Digital I/O
• A/D Converter and more...

Call Today for a Free Catalogue !

Industrial Automation with PCs
ADVANTECH

U.S.A.

750 East Arques Avenue Sunnyvale, CA 94086
Tel: (408)522-4696 Fax: (408)245-8268
E-mail: info@advantech-usa.com

International

FL4, No. 108-3 Ming-Chuan Rd., Shing-Tien, Taipei, Taiwan
Tel: 886-2-2184567 Fax: 886-2-2183875
http://www.advantech.com.tw
E-mail: EBC@acl.advantech.com.tw

CIRCLE # 39 ON READER SERVICE CARD

PID Algorithm

BUMPLESS TRANSFER AND SETPOINT CHANGES

Bumpless transfer is the act of taking a process from manually-controlled to PID-controlled without any noticeable upset or "bump" in the process. I will use my extruder pressure control PID for the next examples in order to drive home this concept.

Let's say my extruder screw is running manually controlled at 15 RPM. If I were to use the above code alone and put the screw in auto (PID loop on), the output of the loop on the first execution would go from 15 RPM in manual to something barely above zero, based on the pressure error at the instant the loop executed. The output would then head off towards the setpoint at a rate dependent upon gain values. Not very bumpless. In order to make the transition bumpless, I need to somehow have the output of the PID loop equal the manual output at the instant of manual to auto switchover. I do this by "pre-loading" the Iterm with the manual output command on the first execution of the PID. The loop is then kept in an integral-only mode until the PV meets the SP, and then it is switched into PID mode.

Smoothing out setpoint changes will allow the operator to make large setpoint changes without upsetting the process. This outcome is often achieved with a "ramped setpoint." I chose a method which resulted in less code by switching the controller to an integral-only mode until the process reaches the setpoint. The controller is then switched back into PID mode.

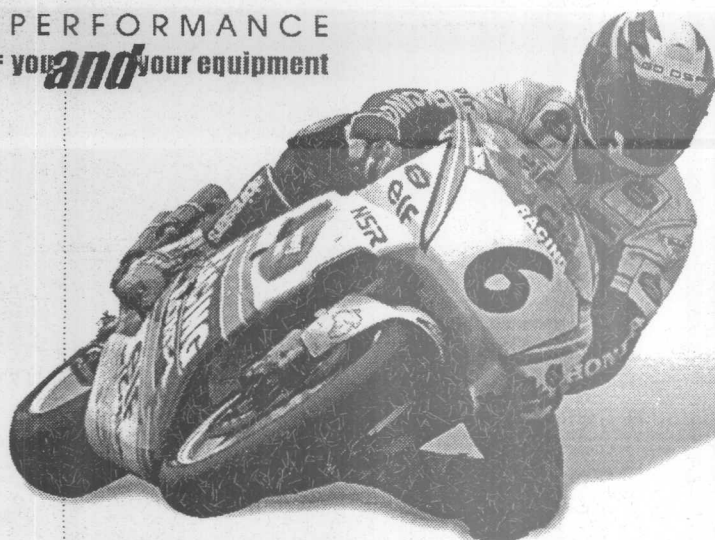
Let me say a few words about Listing 1:

- The PID is set in an infinite loop between Line 01 and Line 53
- It is then broken down into two modes, auto and manual, which are determined by the If-Then-Else statement starting on Line 02, Else at Line 49, and ending at Line 52
- The Auto section is from Line 03 to Line 48 and has two subsections

- The Auto subsection 1 is the bumpless transfer section from Line 04 to Line 22
- The Auto subsection 2 is the normal operation section from Line 24 to Line 46
- Within the normal operation section, there is one more set of subsections
- Normal operation subsection 1 is PID control from Line 25 to Line 31
- Normal operation subsection 2 is for setpoint changes from Line 33 to Line 45

In order to understand how the Listing 1 PID works, let's walk through a typical scenario of startup and transition into Auto or PID mode. Prior to start, the operator makes sure the machine is set in manual mode and then enables the screw servo loop. This action executes the Listing 1 code within the infinite loop (Lines 01 to 53). The operator then enters a `Manual_Command` (RPM) on the operator interface and the screw begins turning at that RPM. The only statements that are performing any servo control at this point are Lines 50 and 51. These lines keep the servo at its manual setpoint and reset the `First_PID_Execution` flag. When polymer pressure is up to desired level, the operator places the screw in Auto (PID mode). From here on the only control code executed will be between Lines 02 and 49. On first scan in PID mode, Line 07 sets up the Iterm "pre-load", Line 08 sets the `First_PID_Execution` flag so these three lines of code will only run once, and Line 09 sets an `Initial_Error` variable to the error. This will be used in conjunction with Lines 14 to 22 to gently bring the extruder pressure up to setpoint. The next three lines of code set the three PID terms and you'll notice that the `Pterm` and `Dterm` are set to zero. This leaves an I-only controller whose Iterm will slowly (gently, bumplessly) get larger or smaller depending on the error until it forces the output to a point where the PV passes the SP. This event is picked up

PERFORMANCE
= your *and* your equipment



CHECK THE WEB

www.**go-dsp**.com
High Performance **DSP Tools**

Open Plug-In Architecture • DSP System Visualization & Management • Debugging • File I/O
Signal Probe Points • Graphical Signal Analysis • Multiprocessing • Interactive Profiling • IDE
Windows-NT • Unix • TMS320C6x

GO DSP Corp. tel: (+1) 416-599-6868

CIRCLE # 41 ON READER SERVICE CARD



```
02A4      x40;  
02A6      s&0x20)  
02A9      table();
```

Optimized C for Embedded Applications

Byte Craft's optimizing C compilers are fast and efficient. C extensions provide control over bit manipulations, I/O port and memory definitions, as well as support for direct register access and interrupts.

We respond to your C compiler needs.

PIC16/17CXX
MELPS740
MC68HC05
MC68HC08
COP8
Z8

Byte Craft Limited

421 King Street North
Waterloo, ON CANADA
N2J 4E4

Tel: (519) 888-6911

Fax: (519) 746-6751

BBS: (519) 888-7626

Email: info@bytecraft.com

www.bytecraft.com

CIRCLE # 42 ON READER SERVICE CARD

PID Algorithm

by either Lines 14 to 17 or 18 to 2 depending on the sign of the Initial_Error and the controller is then shifted into PID mode from Lines 25 to 31. Before we leave the bumpy transfer code explanation, note Line 16 and 20—Last_PV is set equal to PV. This is necessary to assure that on first calculation of the Dterm, its output is stable. Otherwise a severe "bump" may result on transition.

Lines 25 to 31 should be self-explanatory by now for they are the standard, no-frills PID. If the operator should select a new pressure setpoint (SP) code from Lines 33 to 45 execute until the PV passes the new SP. You'll notice that this code looks similar to the bumpless code mentioned above. The code uses the same technique of turning the PID into a straight I-controller, which is more gentle about seeking a large setpoint change than a full PID. This section of code is executed until either Line 38 or 41 (depending on the relationship between SP and Last_SP) see that the SP has been passed by the PV. It is then shifted back to standard PID.

SMOOTH OPERATOR

As you can see, PID code alone is fairly simple. The trickiest part comes in when trying to build a PID that operates smoothly in all modes of operation. Many ways exist to accomplish PID control, bumpless transfer, and handle setpoint changes. Other methods that could and probably should be tried are variable gains and ramped setpoint. Each method must be weighed against the type of PID algorithm used. What worked on a parallel algorithm may not work well on an ISA PID. **ESP**

Garth Gaddy has been working in the embedded control field for five years and his company is G2 Data and Control. He is a licensed pilot and A&P mechanic, and he's currently working on a computerized flight control system and auto pilot for home built aircraft. Garth can be reached at grg01@alumni.csufresno.edu.